A Framework For Analysing The Effect Of 'Change' In Legacy Code

Shikun Zhou Hussein Zedan* Antonio Cau

Software Technology Research Laboratory
SERCentre, De Montfort University,
The Gateway, Leicester LE1 9BH, England
E-mail: {kzhou,hzedan,acau}@strl.dmu.ac.uk

Abstract

We propose a sound and practical approach, based on a formal method (known as Interval Temporal Logic), to cope with 'change' and analyse its effect. The approach allows us to capture a snapshot of system's behaviour over which various interesting properties, such as liveness, timeliness and safety properties, can be validated **compositionally**. *These properties may include invariants that are required to be valid after changes have taken place. We also present and evaluate the design and implementation of a formal tool,* **AnaTempura**, *which supports the developed approach. A case study is presented to illustrate our approach and the tool.*

1. Introduction

Computing systems, both hardware and software, are continually evolving. This evolution will inevitably lead to their rapid growth in size and change to their original requirements rendering them to 'legacy' status. In fact, some consider a system to be in a legacy state even before it is being deployed!

The evolution of software system could be due to changes in the original requirements, adopting a different hardware platform or to improve its efficiency. The classification of maintenance approach [15] indicates the various types of change that can be seen through the maintenance process. As a result, this is seen as an indication of the type of evolutionary changes that may occur to software. Because of its complexity, the likelihood of subtle errors is much greater and some of these errors could have catastrophic consequences such as loss of life, money, time or damage to the environment. Therefore managing *system evolution* is a crucial aspect in system development and maintenance.

Hence, a fundamental issue that faces a software maintainer is how to response to '*change*'. This response must be undertaken rapidly, efficiently and, above all, correctly. As the maintainers are not usually the original developers of the system, responding to changes requires understanding the system, identify the necessary changes and then perform the changes.

For *critical* applications, using formal methods, which are mathematically-based techniques, is fundamental as they greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected [27, 16].

For large scale systems, comprehension, by necessity, will be partial. Corrective maintenance is regarded as the most common activity during the life time of the program. 'Hypotheses', at the code, algorithmic and application domain levels, are major drivers to program understanding in the corrective maintenance activities [24, 4, 14]. Program slicing, both static [25, 8] and dynamic [13, 10, 1], is also a technique often used in maintenance activities such as comprehension, design recovery and risk migration. The technique was further used to identify functionalities [9]. Determining slicing criteria [7] was achieved using symbolic execution [11].

Another important issue in managing change is to establish mechanisms to cope with its propagation. The change is often made to a specific part of the system. After the change, that part may no longer be compatible with other parts of the system, as it may no longer provide what was originally expected or it may now require different services for the rest of the system. These dependencies need to be checked, validated and re-established if they are lost. The process in which the change spreads through the software is sometimes called the ripple effect of change [28]. Various techniques have been proposed to model change [2, 3] and and its impact [21, 20, 19]. The prediction of the size and location of change has also been considered (e.g. [8]).

*The author wishes to acknowledge the funding received from the U.K. Engineering and Physical Sciences Research Council (EPSRC) through the Research Grant GR/M/02583

Data clustering techniques [26] were used [5] to study the process of software evolution, focusing on calling structure and data used within (and externally accessible) the application.

This paper aims to present a sound technique, together with its supporting tool, for handling *continuous* change in system development and maintenance. Using our technique, we can validate and analyse system's behaviours of interest. The validation and analysis are performed within a *single* logical framework using Interval Temporal Logic (ITL [17, 6]) and its executable subset, Tempura [18]. Behavioral properties such as safety, liveness, timeliness, are expressed in ITL as theorems which can then be validated and tested *compositionally*. These properties include invariants that may be required to be valid before and after the change is made.

The technique presented here is language independent. The source code could be in any language (e.g. C, C++, Ada and COBOL). It is also suitable for concurrent/distributed legacy systems. The adoption of ITL allows our technique to be used for the temporal/timing analysis in the reengineering process. The technique can also be used to increase comprehension about the system.

The paper is organised as follows. Section 2 introduces our formal model. Section 3 describes the integrated framework, and Section 4 describes the design and implementation of the analysing tool AnaTempura. Section 5 is devoted to several case studies to illustrate the approach and its associated tool support. We conclude in Section 6 with some remarks and future work.

2. Preliminaries

2.1. ITL Syntax

Interval Temporal Logic (ITL) is a flexible notation for both propositional and first order reasoning about periods of time found in descriptions of hardware and software systems. It can handle both sequential and parallel composition unlike most temporal logics. It offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and timeliness.

	Expressions
$e ::=$	$\mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \imath a{:}\, f$
	Formulae
$f ::=$	$p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid$ $\forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$

Figure 1. Syntax of ITL

The syntax of ITL is defined in Fig. 1 where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol. An interval is a sequence of states.

The informal semantics of the most interesting constructs are as follows:

- $\imath a{:}\, f$: the value of a such that f holds.
- skip: unit interval (length 1).
- $f_1 ; f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.
- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

The following are a few examples illustrating ITL:

In an interval, the variable I at *some time* equals 1 and at *some later time* equals 2 can be expressed as:

$$\Diamond[(I = 1) \wedge \Diamond(I = 2)]$$

In an interval, if the variable I *always* equals 1 and in the next state the variable J equals 2 then it follows that the expression I + J equals 3 in the next state:

$$\models \quad [\Box(I = 1) \wedge \bigcirc(J = 2)] \quad \supset \quad \bigcirc(I + J = 3)$$

The formula $(I + 1 \rightarrow I) ; (I + 2 \rightarrow I)$ is true in an interval if and only if that interval can be chopped into two subintervals such that the sub-formula $I + 1 \rightarrow I$ is true on the first subinterval and the sub-formula $I + 2 \rightarrow I$ is true on the second subinterval. The net effect is that I increases by 3. This is expressed by the following property:

$$\models \ [(I + 1 \rightarrow I) ; (I + 2 \rightarrow I)] \ \supset \ (I + 3 \rightarrow I)$$

These constructs enables us to define programming constructs like assignment, if then else, while loop etc. Appendix A contains some frequently used abbreviations.

2.2. Types in ITL

Introducing a type system into specification languages has its advantages and disadvantages. An untyped set theory is simple and is more flexible than any simple typed formalism. Polymorphism, overloading and sub-typing can make a type system more powerful but at the cost of increased complexity. While types serve little purpose in hand proofs, they do help with mechanised proofs.

There are two basic builtin types in ITL (which can be given pure set-theoretic definitions). These are integers $\mathcal{N}$ (together with standard relations of inequality and quality) and Boolean (*true* and *false*).

Further types can be built from these by means of $\times$ and the power set operator $\mathcal{P}$ (in a similar fashion as adopted in the specification language Z).

For example, the following introduces a variable x of type T

$$(\exists x : T) \cdot f \;\; \widehat{=} \;\; \exists x \cdot (type(T) \wedge f)$$

Here $type(T)$ denotes a formula describing the desired type. For example, $type(T)$ could be $0 \leq x \leq 7$ and so on. Although this might seem to be a rather inexpressive type system, richer types can be added.

2.3. Tempura

Our choice of ITL in this work is based on the availability of an executable subset of the logic. This offers a flexible and rapid prototyping system, known as Tempura. Its syntax resembles that of ITL. It has as data-structures integers and booleans and the list construct to built more complex ones.

The standard operations on expressions are available like $+, -, *, /, div, mod, =, >, or, and$. The basic statements (with the corresponding ITL construct) are as follows: (for more details, we refer the reader to [18]):

ITL	$Tempura$
$f_1 \wedge f_2$	f_1 *and* f_2
$A := exp$	$A := exp$
more	*more*
empty	empty
$\Diamond$	*sometimes*
$\Box$	*always*
ⓦ	*wnext*
true	*true*
false	*false*
if b then f_1 else f_2	if b then f_1 else f_2
while b do f	while b do f
repeat b until f	repeat b until f
"procedures"	*define* $p(e_1, \ldots, e_n) = f$
"functions"	*define* $g(e_1, \ldots, e_n) = e$

An example is as follows:

```
define test() = {
  exists X,Z : {
    if fc>1 then {
     prog_send(fc); skip;
     {len(fc-2) and  Z=0 and
      always (input X and check(X,fac(Z))) and
      Z gets Z+1 }
    } else { prog_send(fc) }
  }
}.
```

Some of the main constructs are used in this example. The "define" defines the procedure of "test()". The "exists" operator denotes existential quantification and corresponds to the introduction of local variables, "X" and "Z". As used in normal languages, the notation "if ... then ... else ..." represents a binary choice. The predicate "skip" specifies an interval of length one. The "len" construct is used to define an interval length "(fc-2)". The statement "Z=0" initialise the variable "Z". The operator "and" is used to compose formulae in parallel. The construct ";" means sequential composition. The operator "always" causes the statement "(input X and check(X,fac(Z)))" to be executed on every state. The "input" is used to read a value externally and put it to the local variable "X". The operator "gets" indicate that the value of "Z" on the next state is always equal to the value of "Z+1" on the current state. The rest of constructs, "prog_send(fc)" and "check(X,fac(Z))", are all specially defined functions.

2.4. The Tempura Interpreter

The Tempura interpreter is used to execute the various constructs of Tempura. Here we just give a brief description how the interpreter works with help of an example, more details can be seen in [18].

Towards a Tempura program:

$$(next\ next\ empty)\ and\ (I = 0)\ and$$
$$(I\ gets\ I + 1)\ and\ always\ (J = 2 * I)$$

This program is simple enough to make a mental calculation and come to the conclusion that this is true on intervals of length 2 in which "I" assumes the value 0, 1 and 2 while "J" simultaneously assumes the values 0, 2 and 4. One way to execute such a formula is to transform it to a logically equivalent conjunction of the two formulas $present\_state$ and $wnext\ what\_remains$[1]:

$$present\_state\ and\ wnext\ what\_remains$$

Here, the formula $present\_state$ consists of assignments to the program variables and also indicates whether or not the interval is finished. The formula $what\_remains$ is what is executed in subsequent states if the interval does indeed continue on. Thus, it can be viewed as a kind of continuation. For the formula under consideration, $present\_state$ has the following value:

$$(I = 0)\ and\ (J = 0)\ and\ more.$$

The value of $what\_remains$ is the formula:

$$(next\ empty)\ and\ (I = 1)\ and\ (I\ gets\ I + 1)$$
$$and\ always\ (J = 2 * I)$$

Below we also show the effects of such transformations before and after each of the three states of the execution.

Before state 0:

$$(next\ next\ empty)\ and\ (I = 0)\ and$$
$$(I\ gets\ I + 1)\ and\ always\ (J = 2 * I)$$

[1] In order for the construct $wnext\ w$ to be true on an interval, either the interval is empty or the sub-formula w is true in the next state

After state 0:

$$[(I = 0)\ and\ (J = 0)\ and\ more]\ and$$
$$wnext\ [(next\ empty)\ and\ (I = 1)\ and$$
$$(I\ gets\ I + 1)\ and\ always(J = 2 * I)]$$

Before state 1:

$$(next\ empty)\ and\ (I = 1)\ and\ (I\ gets\ I + 1)$$
$$and\ always(J = 2 * I)$$

After state 1:

$$[(I = 1)\ and\ (J = 2)\ and\ more]\ and$$
$$wnext\ [empty\ and\ (I = 2)\ and$$
$$(I\ gets\ I + 1)\ and\ always(J = 2 * I)]$$

Before state 2:

$$empty\ and\ (I = 2)\ and\ (I\ gets\ I + 1)$$
$$and\ always(J = 2 * I)$$

After state 2:

$$[(I = 2)\ and\ (J = 4)\ and\ empty]\ and$$
$$wnext\ [false\ and\ (I\ gets\ I + 1)$$
$$and\ always(J = 2 * I)]$$

3. The Approach

In this section we describe our approach. We begin by establishing a computational model which is suitable for modeling legacy system albeit sequential or concurrent, timed or untimed.

3.1. Computation

We take the view that a computation defines mathematically an abstract architecture upon which applications will execute. A legacy *system* is a collection of *agents* (which is our unit of computation), possibly executing concurrently and communicating (a)synchronously via communication links. Systems can themselves be viewed as single agents and composed into larger systems. Systems may have timing constraints imposed at three levels; system wide communication deadlines, agent deadlines and sub-computation deadlines (within the computation of an individual agent).

At any instant in time a system can be thought of as having an unique *state*. The system state is defined by the state variables of the system and, for concurrent system, by the values in the communication links, the so called *frame*. This frame defines the variables that can possibly change during system executing, the variables outside this frame will certainly not change. *Computation* is defined as any process that results in a change of system state. An agent is described by a computation which may transform a private data-space and may read and write to communication links during execution. The computation may have both minimum and maximum execution times imposed.

A local data-space for the agent is created when an agent starts execution with initial values which are nondeterministic. The data-space is destroyed when the agent terminates. No agent may read or write another agent's data-space[2].

3.2. Behaviours and Properties

An interval is considered to be a (in)finite sequence of states, where a state is a mapping from the set of variables to the set of values. The length of an interval is equal to one less than the number of states in the interval, i.e., a one state interval has length 0.

A *behaviour* in our model is defined as a sequence of states, i.e., an interval in ITL. Hence, a behaviour could be finite or infinite. A behaviour is called *full* behaviour if it contains all the state variables of the system otherwise it is called *partial*. A partial behaviour can be obtained by hiding some state variables (formally it is a projected behaviour over state variables).

A property P can be either a *state* or *temporal* ITL formula, i.e., a set of behaviours. A general classification of properties are readily available: **safety** (*something bad does not happen*) and **liveness** (*something good will eventually happen*) property. Another class of properties are known as *temporal/timing* (*something happens within a duration of, by or at a certain time deadline*).

3.3. Assertion points

A mechanism for managing change in a legacy system should be practical, systematic and compositional. A fundamental issue in our approach is the ability to capture the behaviour of (sub-)system. Once the behaviour is captured then we can assert if such behaviour satisfies a given property. And as a property is a set of behaviours, *satisfaction* is achieved by checking if the captured system's behaviour is an element of this set. We are not dealing here with the formal verification of properties which requires that all possible behaviours of a system satisfy the properties. The formal verification of these properties may also be performed using an ITL verifier. We are only concerned with validating properties which requires that only interesting behaviours satisfy the properties.

The states of a (sub-)system to be changed are captured by inserting *assertion points* at suitably chosen places. These divide the system into several *code-chunks*, as depicted in figure 2. Properties of interests are then validated over this behaviour.

[2]For concurrent/distributed legacy applications, we assume that an agent may write to at most a finite number of communication links and read from at most a finite number of them. Synchronous communication links, i.e., read and write occur at the same time, are called *channels*. Asynchronous communication links are called *shunts*. Shunt writing is destructive, shunt reading is not.

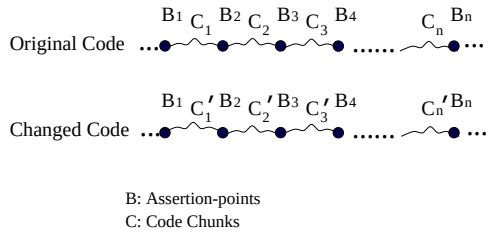

Figure 2. Assertion points and Chunks

Our general framework can be systematically described as follows.

1. Establish all desirable properties of the system under consideration and express them in Tempura.
2. identify suitable places in the legacy code and insert assertional points.
3. Using Tempura, check that the behaviour of the legacy system satisfies the desired properties.

Establishing system properties can be a hard task, however we suggest to follow the main characterisation of properties given above, namely safety, liveness and timing properties. Obviously, some level of understanding of the (sub-)system under consideration is assumed. These properties could be invariants that need to be hold true through the system and/or after the changes are made.

The locations of assertion points could be chosen, for example, at the entry and exit points of a procedure or function. In this case assertions are in fact *pre-* and *post-* conditions, and what we are asserting is: If the system starts at a state satisfying the *pre-* condition then it terminates properly in a state satisfying the *post-* condition.

In addition, *assumption/commitment* style properties [29] could be used: For a system Sys the *assumption/commitment* style can be expressed in ITL as follows:

$$\vdash\ w \wedge As \wedge Sys\ \supset\ Co \wedge \mathit{fin}\ w'.$$

This states that if the state formula w is true in the initial state and the assumption As is true over the interval in which Sys is operating, then the commitment Co is also achieved. Furthermore the state formula w' is true in the interval's final state or is vacuously true if the interval does not terminate. This is particularly important as As could be a formula asserting various assumptions about the environment in which the system, under consideration, is operating.

If a change occurs in a code chunk, say C_2 in figure 2 to produce C'_2, we are now able to check the ripple effect of such a change on C_i and $i \neq 2$.

Moreover, we can use the *assumption/commitment* technique *forcing* changes made to have no or a desirable effect on the neighbouring code chunks. This is depicted in Figure 3 in which a new code chunk is to be inserted at P. The environment of this new addition, code chunk C, consists of at least A' and B'. The development of C could be controlled in a such way that, for example $A' = A$.

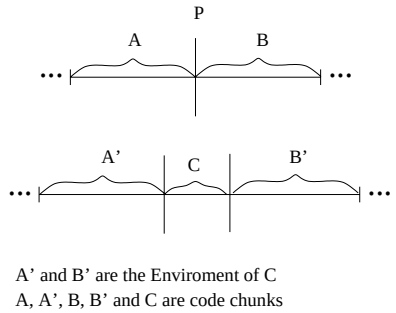

Figure 3. Environment

Within our framework we can *compositionally* validate properties. For example the following rule for sequential composition is sound

$$\frac{\vdash\ w \wedge As \wedge Sys\ \supset\ Co \wedge \mathit{fin}\ w' \quad \vdash\ w' \wedge As \wedge Sys'\ \supset\ Co \wedge \mathit{fin}\ w''}{\vdash\ w \wedge As \wedge (Sys; Sys')\ \supset\ Co \wedge \mathit{fin}\ w''\ .} \tag{1}$$

Similar rules for iteration and concurrent operators are also available.

4. Tool Support

We have designed and implemented a tool, known as *AnaTempura*, that support the approach described above. In this section we describe the design and its use via a simple working example written in C to calculate the factorial. The text of the program is depicted in Figure 4.

```
#include <stdio.h>
/* Factorial tester */
main()
{
int y, fac=1;
 printf ("Enter the seed: \n");
 scanf ("%d", &y);
 while (y>0) {
  fac=1;
  while (y>1) {
   fac=fac*y; y=y-1;
   printf("PROG: assert fac:d:%d::\n",fac); (*)
  }
  printf ("Enter the seed: ");
  scanf ("%d", &y);
 }
 printf ("PROG: end ::\n");
}
```

Figure 4. The C Code of the Factorial

Figures 5 and 6 show the general structure and data flow of the tool, respectively. There are two main parts: A *Monitor* and the *Tempura Interpreter*. The Monitor acts as interface between the legacy code and the Tempura Interpreter.

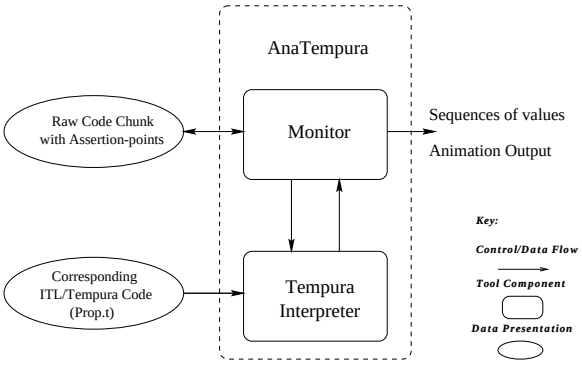

Figure 5. Basic Functions

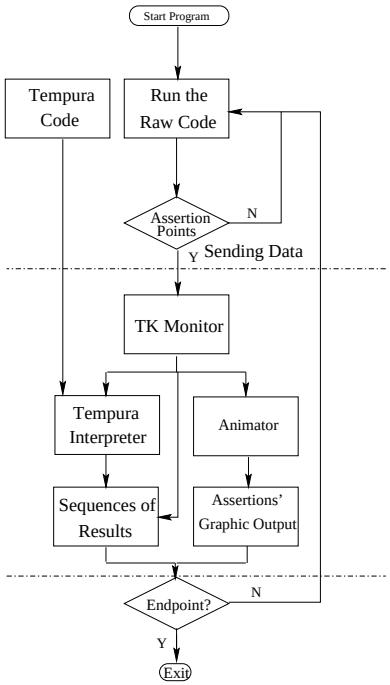

Figure 6. Control Structure of the Technique

The starting point is formulating, in Tempura, all behavioral properties of interest, such as safety, liveness and timeliness. these are stored in a Tempura file ("Prop.t", Figure 7 in our case). In our working example, the safety property, i.e. *nothing wrong happens*, is

$$always\ (input\ X\ and\ check(X, fac(Z)))$$

where `fac(Z)` is shown in Figure 7.

The assertion-point has been inserted into the body of the C code (Marked by "(*)" in Figure 4). In this case,

```
define fc = 7.
define check(X,Y) = {
  if strint(suf(X,6))= Y then {
    format("Pass test\n")
  } else { format("Fac: Prog %d Real %d\n",
           strint(suf(X,6)),Y) }
}.
define fac(K) = {
   if K=0 then fc else (fc - K) * fac(K-1)
}.
define test() = {
  exists X,Z : {
    if fc>1 then {
     prog_send(fc); skip;
     {len(fc-2) and  Z=0 and
      always (input X and check(X,fac(Z))) and
      Z gets Z+1}
    } else { prog_send(fc) }
  }
}.
```

Figure 7. The Tempura Code of the Factorial

we have just one assertion-point corresponding to the only safety property. This assertion-point will send values of `fac` during the computation of the factorial. This sequence will be the behaviour of the system and for which we check the satisfaction of our property. The safety property being checked is that the loop for computing the factorial satisfies a certain invariant.

This checking is done as follows: start the *AnaTempura* and load the Tempura program. *AnaTempura* will then start the corresponding legacy system, in our case the factorial program (compiled C code chunk with assertion-points). The legacy system will then generate a behaviour via its assertion points. We then start the Tempura program to check that this behaviour satisfies the properties. The results of validating the factorial program are in Figure 8. There is also an facility to animate the behaviours received from the legacy system.

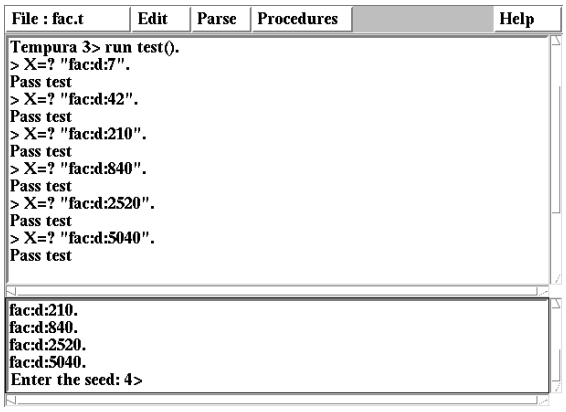

Figure 8. Validation Results

We note here that if the properties are not satisfied, AnaTempura will indicate the errors by displaying what is expected and what the legacy system actually provides. Therefore, the approach is not just a "keep-tracking" approach, i.e. giving the running results of the interesting properties of the legacy system. By not only capturing the running results but also comparing them with the formalised properties, the AnaTempura tool can validate the interesting properties.

5. Case Study

5.1. Description

The system is used to sort First and Second class letters and put them into different hoppers as illustrated in the left hand side of Figure 11. The size of the main controlling program in C is some 220 lines of code. The size of the total system is some 2.5K lines of code. A class sensor detects the different colours of stamps and send corresponding signals to the control program which in turns sends signals to the switch (solenoid). The switch will turn the sorting gate *on* or *off* so as to release two classes of letters into different hoppers.

Delays in the system occur at different places as shown in the left hand side of Figure 11: Delay 4 (75ms) and Delay F (250ms) for Solenoid 4 and Letter Sensor respectively; Delay 3A (75ms) and Delay 3B (75ms) for the Solenoid 3 [3].

The sorter system is required to be extended in such a way that *Air Mail* letters are also sorted.

5.2. The current system

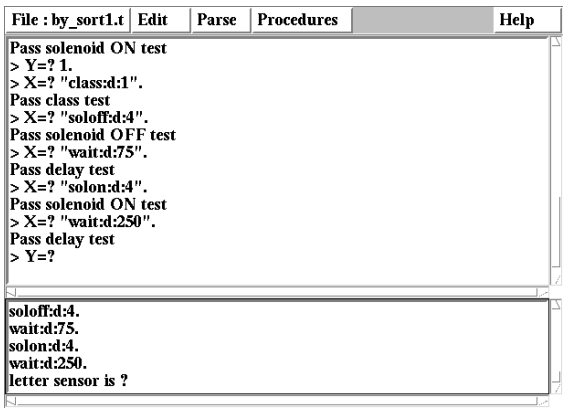

Figure 9. Validation of Sorting 1st/2nd Class

We begin by analysing the current system. Two properties are of interest: timeliness and safety properties. The

[3]These variable names come from the original code

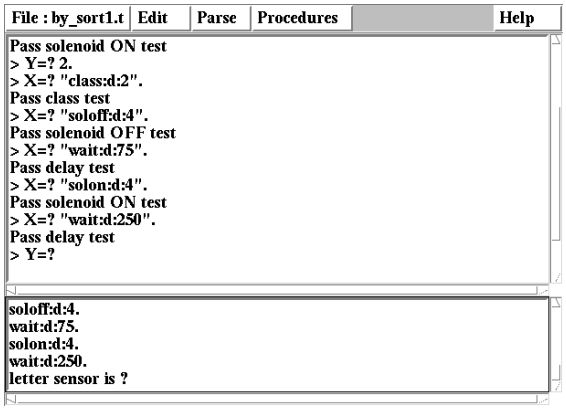

Figure 10. Validation of Sorting 2nd Class

former involves the delay times for switching the solenoids and reading the sensors, whilst the later is that *no Second class letter drops in the first class hopper and visa versa*. These will have to be validated before the required changes are made.

These properties are whether the Class and Letter Sensors indicate the correct states of letters and switches (Solenoid 3 and Solenoid 4), and whether the various time delays (Delay 4 and Delay F for Solenoid 4 and Letter Sensor, Delay 3A and Delay 3B for Solenoid 3) are correct. We formulate these properties in Tempura: one property is for sorting letters into the First class hopper and another is for sorting letters into the Second class hopper.

Due to the limitation of the space, we have not presented the whole ITL and Tempura code in this paper. Instead, we give an example formula which is used to define one of the main timing properties.

The Class Sensor should detect the class of a letter and send the corresponding signal to the controller within 75 time units. This unit of computation will be defined as the following *agent*.

$$Cs\_controller \mathrel{\widehat{=}} [75]\ Cs\_check\_send$$

where

$$[t]\ S \mathrel{\widehat{=}} len = t \wedge (S\,;true) \wedge (S\ \supset\ len \leq t)$$

i.e., if checking the class and sending of the signal is less than 75 time units then we wait till we reached 75 and if it takes more than 75 time units then generate an error. Each bit of the controller can be described in such a way.

Two test runs are shown in Figures 9 and 10. AnaTempura runs the program, shown by "run test()" on the top half of figures. The bottom half of the figures serves as a terminal to show outputs and inputs of the running program. Assertion-points are placed strategically in the code that are most appropriate to the safety property and to the timeliness property. In particular, the results of "class:d:1",

"soloff:d:4" and "solon:d:4" correspond to the safety property of the program, i.e., whether a First class can be released into the First class hopper. Meanwhile, the results of "wait:d:75" and "wait:d:250" correspond to the timeliness property, i.e., whether the program can get the signal from sensors and send control signals to the actuators in time.

The Tempura Interpreter checks the received behaviour corresponding to the assertion-points of the running program, and give appropriate messages like "Pass solenoid ON test" and "Pass delay test", which indicate that both the safety and the timeliness properties are satisfied by the raw code.

5.3. The new system

The chosen properties of the current system have now been validated. This gives the maintainer enough confidence to carry out the required changes. (More properties could be validated to increase assurance.)

In this section, we carry out the required changes to the system so as to satisfy the new requirements, i.e., *sorting Air Mail letters*. Further we show how such changes affect the system. What is required is that the new system should at least satisfy both safety and timeliness properties given above. In fact the new system will satisfy extra properties.

The new structure can be seen in the right hand side of Figure 11. A new "Air Mail Sensor" and two new actuators (Solenoid 6 and Solenoid 5) were added. This has introduced new delays (Delay 5A, Delay 5B and Delay Fa). The C code has been modified to cater for the new addition.

The observant reader will notice that the structure of the new system (right hand side of Figure 11) has changed to cater for the new addition and also make it more efficient: The Letter Sensor has been moved forwards, i.e., from the position between Solenoid 4 and Solenoid 3 to the new position between Solenoid 4 and Solenoid 5.

We have modified the original Tempura code. We have also added corresponding assertion-points to the newly added items. Three assertion-points corresponding to the new sensors and actuators relate to the modified safety property, while the three other assertion-points corresponding to the new delays relate to the timeliness property. Then we use the same procedure to validate the newly added properties, i.e., the safety property and timeliness property that each Air Mail letter should be delivered into the air mail hopper within a certain time.

We describe the validating process of the new additions. The results are presented in Figure 12. They indicate that the safety property has been met for this particular run. The program has sent correct control signals to the actuators, Solenoid 4 and Solenoid 5, and all Air Mail letters have been delivered to the Air Mail Hopper. At the same time, all timeliness properties have been also satisfied. The delay times for the sensors and solenoids are correct.

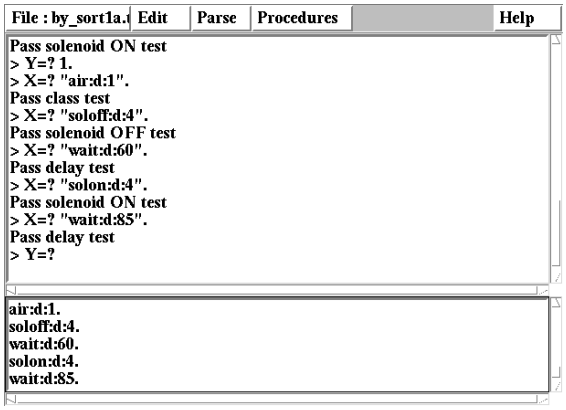

Figure 12. Validation of Sorting Air Mail

6. Conclusion

We have presented a sound and practical approach, based on a formalism known as Interval Temporal Logic, to manage 'change' in legacy system. The approach enables the maintainer to analyse the effect of change. It allows us to capture a snapshot of system's behaviour over which various interesting properties, such as liveness, timeliness and safety properties, can be validated compositionally. These properties could be invariants that are required to be valid after changes have taken place.

We also presented the supporting tool known as **AnaTempura**. The tool was used on a small but illustrative case study of a mail sorter.

Among the first analysers are Anna [22] and PLEASE [23]. Anna is a language extension of Ada to include facilities for formally specifying the intended behaviour of Ada programs. It augments Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs. Like Anna, PLEASE allows software to be annotated with formulae written in predicate logic; annotations can be used in proofs of correctness and to generate run-time assertion checks. As the logic in PLEASE is restricted to Horn clauses, specifications can be also transformed into prototypes which use Prolog to 'execute' pre- and post-conditions. Anna and PLEASE however do not deal with timing properties.

We believe that one of the strengths of our approach is its sound foundation. Interval Temporal Logic (ITL), with its rich axiomatic system and compositional proof rules, enables properties of interests to be verified and validated over an interval of time. The Assumption/Commitment-style proof rules gives a powerful tool to force changes to

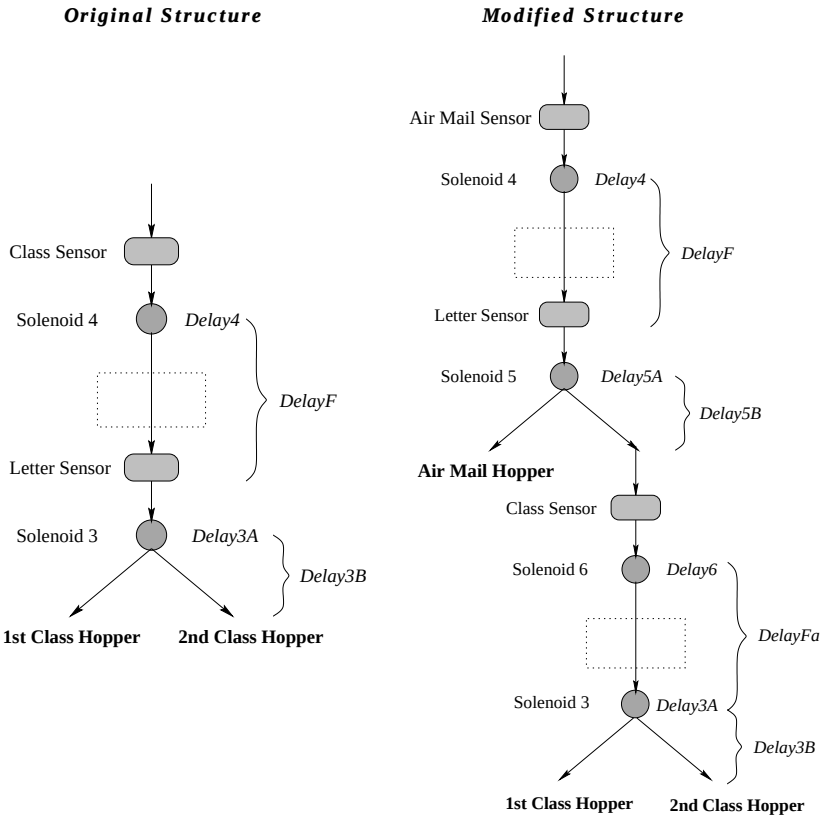

Figure 11. The Structure of the Original and Modified Mail Sorter

have either *no* or *desirable* effects on the environment of the system under consideration.

Unlike other tools and techniques, **AnaTempura** can be used for both sequential and parallel systems. In addition the application domains cover both timed and untimed applications. It is language independent, i.e., it can be used to process the legacy code in any language, such as C, C++, Ada and COBOL.

As future work, we plan to integrate the **AnaTempura** with an ITL verifier/Proof Checker, such as Lite [12], which offers the possibility to verify that *all possible* behaviours of the legacy system satisfy a given property. The tool will ultimately be integrated with the Re-engineering Assistant (RA) [27]. Currently, we are undertaken larger case studies from various application domains, including manufacturing industry and finance. Our experience of this study will be reported elsewhere.

Establishing assertion points is currently done manually and relies on a good level of understanding of the system. We aim to develop mechanisms/guidelines for the establishment and insertion of assertion-points, e.g., the location and the number of assertion-points. Also we will use the *assumption/commitment* technique introduced in section 3.3 to determine our assertion points.

We currently working on visual notation for ITL which will be incorporated within **AnaTempura**. This will enhance the acceptability of ITL to non-experts in the specification of system properties.

Acknowledgement

The authors wish to thank colleagues at the Software Technology Research Laboratory for their helpful criticisms and discussion during this work.

References

[1] H. Agrawal and J. Horgan. Dynamic Program Slicing. *Proc. of the ACM SIGPLAN'90 Conf of Programing Lang. Design and Implementation*, 1990.

[2] R. Arnold and S. Bohner. Impact Analysis - Towards a Framework for Comparison. *Proc. Int. Conf. Software Maintenance*, 1993.

[3] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[4] R. Brooks. Towards A Theory of the Comprehension of Computer Programs. *Intr. Journal of Man-Machine Studies*, 18, 1983.

[5] E. Burd and M. Munro. Investigating Component-Based Maintenance and the Effect of Software Evolution: A Re-engineering Approach Using Data Clustering. *ICSM98*, 1998.

[6] A. Cau and H. Zedan. Refining Interval Temporal Logic Specifications. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development*, number 1231 in LNCS, pages 79–94. AMAST, Springer-Verlag, 1997.

[7] A. Cimitile, A. D. Lucia, and M. Munro. Identifying Reusable Functions Using Specification Driven Program Slicing. *Proc of ICSM'95*, 1995.

[8] K. Gallagher and J. Lyle. Using Program Slicing in Software Maintenance. *IEEE TSE*, 17, 1991.

[9] G. Ganfora, A. D. Lucia, G. D. Lucca, and A. Fasolino. Slicing Large Programs to Isolate Reusable Functions. *Proc. of EUROMICRO Conf*, 1994.

[10] M. Kamkar and P. Krajina. Dynamic Slicing of Distributed Programs. *Proc. of ICSM 1995*, 1995.

[11] J. King. Symbolic Execution and Program Testing. *CACM*, 9, 1976.

[12] S. Kono. Automatic Verification of Interval Temporal Logic. In *Proc. of 8th British Colloquium For Theoretical Computer Science*, March 1992.

[13] B. Korel and J. Laski. Dynamic Slicing of Computer Program. *The Journal of Systems and Software*, 1990.

[14] S. Letovsky. *Emperical Studies of Programmers*, chapter Cognitive Processes in Program comprehension. Ablex, 1986.

[15] B. Lientz and E. Swanson. *Software Maintenance Management*. Addison Wesley, 1980.

[16] X. Liu, H. Yang, and H. Zedan. Formal Methods for the Re-engineering of Computing Systems. In *Proceedings of The 21st IEEE International Conference on Computer Software and Application (COMPSAC'97)*, pages 409–141, Washington, D. C., August 1997. IEEE Computer Society.

[17] B. Moszkowski. *A Temporal Logic for Multilevel Reasoning about Hardware*. IEEE Computer Society, Feb. 1985.

[18] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge UK, 1986.

[19] J. Munson and S. Elbaum. Code Churn: A Measure for Estimating the Impact of Code Change. *ICSM98*, 1998.

[20] V. Rajlich. A Model for Change Propagation Based on Graph Rewriting. *Proc of ICSM'97*, 1997.

[21] V. Rajlich, N. Damaskinos, P. Linos, and W. Khorshid. VIFOR: A tool for software maintenance. *Software Practice and Experience*, 20, 1990.

[22] S. Sankar, D. Rosenblum, and R. Neff. An Implementation of Anna. In J. G. P. Barnes and J. G. A. Fisher, editors, *Proceedings of the Ada International Conference on Ada in Use*, pages 285–296, Paris, May 1985. ACM, Cambridge University Press.

[23] R. B. Terwilliger. PLEASE: a Language Combining Imperative and Logic Programming. *SIGPlan Notices*, 23(4):103–110, Apr. 1988.

[24] A. von Mayrhauser and A. M. Vans. Hypothesis-Driven Understanding Processes During Corrective Maintenance of Large Scale Software. *ICSM'97*, 1997.

[25] M. Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of An Automatic Program Abstraction Method*. PhD thesis, University of Michigan, 1997.

[26] T. Wiggerts. Using Clustering Algorithms in Legacy System Remodularisation. *Proceedings of the 4th Working Conference on Reverse Engineering*, Oct 6-8 1997.

[27] H. Yang, X. Liu, and H. Zedan. Tackling the Abstraction Problem for Reverse Engineering in A System Re-engineering Approach. *Proc. of IEEE Conference on Software Maintenance, IEEE*, 1998.

[28] S. Yau, R. Nicholl, J. Tsai, and S.Liu. An Integrated Life-cycle Model for Software Maintenance. *IEEE TSE*, 15, 1988.

[29] H. Zedan, A. Cau, and B. Moszkowski. Compositional Modelling: The Formal Perspective. In *Proceedings in the Workshop on Systems Modelling for Business Process Improvement*, pages Chapter 2 pp 19–44. Artech House, 1999.

A. Frequently used abbreviations

next		
$\bigcirc f$	$\widehat{=}$	skip ; f
wnext		
$\textcircled{w} f$	$\widehat{=}$	$\neg \bigcirc \neg f$
non-empty interval		
more	$\widehat{=}$	$\bigcirc$*true*
empty interval		
empty	$\widehat{=}$	$\neg$*more*
infinite interval		
inf	$\widehat{=}$	*true* ; *false*
finite interval		
finite	$\widehat{=}$	$\neg$*inf*
sometimes		
$\Diamond f$	$\widehat{=}$	*finite* ; f
always		
$\Box f$	$\widehat{=}$	$\neg \Diamond \neg f$
if then else		
if f_0 then f_1 else f_2	$\widehat{=}$	$(f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$
final state		
fin f	$\widehat{=}$	$\Box$(empty $\supset$ f)
some subinterval		
$\diamondsuit f$	$\widehat{=}$	*finite* ; f ; *true*
all subintervals		
$\boxdot f$	$\widehat{=}$	$\neg(\diamondsuit \neg f)$
all unit subintervals		
keep f	$\widehat{=}$	$\boxdot$(skip $\supset$ f)
while loop		
while f_0 do f_1	$\widehat{=}$	$(f_0 \wedge f_1)^* \wedge$ *fin* $\neg f_0$
next value		
$\bigcirc exp$	$\widehat{=}$	$\imath a$: $\bigcirc(exp = a)$
end value		
fin exp	$\widehat{=}$	$\imath a$: *fin* $(exp = a)$
assignment		
$A := exp$	$\widehat{=}$	$\bigcirc A = exp$
temporal assignment		
$exp_1 \leftarrow exp_2$	$\widehat{=}$	*finite* $\wedge$ (*fin* exp_1) $= exp_2$
gets		
exp_1 *gets* exp_2	$\widehat{=}$	*keep* $(exp_1 \leftarrow exp_2)$
stability		
stable exp	$\widehat{=}$	exp *gets* exp